File Harvest:
Targeted, Legal Crawling and Downloading of Online Media

UNDERGRADUATE HONORS THESIS

Presented in Partial Fulfillment of the Requirements for

Distinction in the Degree Bachelor of Science

The Ohio State University

By

Chad Sowald

* * * * *

The Ohio State University

2009

Thesis Committee:

Paolo A.G. Sivilotti, Advisor

Prasun Sinha

Approved by

_____

Advisor
Department of Computer
Science & Engineering

Abstract

Today's internet user has a limited amount of time to manually mine the Internet for content such as videos, images, and documents that they want to view. Much of the user's time is wasted overhead: clicking hyperlinks, waiting for pages to load, and actually downloading the content for offline viewing. Therefore, many users would benefit from an application that could automatically crawl and download a large amount of content from the Internet, so that users could browse and further filter the content offline at a much faster speed and without the unnecessary overhead. I have developed a web crawling and downloading program, File Harvest - written in C# and using the .NET framework - that allows the user to quickly configure the web crawling mechanism before starting it. The web crawler functions by following hyperlinks and examining each page it encounters along the way. The user specifies what web pages to crawl, how many levels of hyperlinks to crawl, and what types of content to download. The primary insight of the work is the value of combining crawling and downloading in a single program – something that related efforts have yet to do. The program uses various web page analysis techniques such as HTTP traffic proxying and static analysis of the page HTML to help the user find as much relevant content as possible to download. There are some limitations as to what can be found through crawling, and these limitations are the primary focus of the research going forward. In general, File Harvest can greatly expedite the discovery and downloading of media for users.

Dedication

To my fiancée, Renae, for always supporting me and embracing my work.

Acknowledgments

First and foremost, my advisor, Paul Sivilotti, has been so accommodating, friendly and sagely. He has single-handily mentored me to the point that I have been able to start, review, edit and complete my undergraduate thesis in only eight short weeks. Not only that, but Paul agreed to work with me without a moment's hesitation. I am grateful to be able to work with Paul—who has gone above and beyond his professorially duties. I want to also thank Prasun Sinha, who has taken on the responsibility of being on my thesis committee with very little notice and time to prepare and has done so with grace. Prasun's expertise in Internet technologies is an ideal match to my research. Lastly, I want to thank the Europa research group for allowing me to present my research to the group and for giving me critical feedback and ideas.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 - INTRODUCTION

Section 1: Overview

Recent estimates (March 2008) indicate that there are now 500 exabytes (10^{18} bytes) of data online that are accessible to internet user—that amount grows every day [1].

At the same time, data available shows that in the United States, the average internet user's online leisure time was down, from 18 hours per week in 2007, to 16 hours per week in 2008 [2] and looks to be stagnant or increasing only slightly in 2009 [3]. This means that the typical user will not be able to keep pace with the dramatic growth rate in available online media. Of course, users do not want to see all online media. Nonetheless, the gap in available online media that users want to view and what they have time and are able to view is increasing.

File Harvest, a Windows application written in C#, automatically discovers and downloads large amounts of online media so that users do not have to manually navigate web pages and download the media themselves. File Harvest helps to decrease the gap in what media users want to view and what they have time to view by eliminating excess manual work. File Harvest still requires some user interaction during setup and configuration, but this configuration time is small in comparison to manually crawling and downloading the same content for sets of media of reasonable size. By automatically crawling and downloading a large amount of content from the Internet, File Harvest allows users to browse and further filter the content offline at a much faster speed. This media could be video, audio, images, documents or any other file type the user specifies.

File Harvest is composed of two main components: a web page crawler and a file downloader. The web page crawler's purpose is to do the work of navigating around web pages, searching for media on those web pages, and recording the location of any media that is found. The file downloader's purpose is simply to download any media that was identified by the crawler to the user's computer.

From a user interaction standpoint, File Harvest can be best viewed as a background application with which the user interacts minimally. After initiating a crawling and downloading session, the user can just let the application run unattended. The configuration and initiation of a session presents some difficulties. Users who are more familiar with URL structure and the general structure of websites will be better equipped to properly configure the application to find the media for which they are looking. Furthermore, they can more easily identify opportunities to use the application in the first place. The simplification of the user interface and the knowledge required by the user is a primary concern of the application's development in future work.

Section 2: Contributions of this Thesis

File Harvest's main contribution is two-fold. First, no other application to date has combined such a sophisticated level of crawling with downloading as well. Instead, most applications tend to focus on one aspect or the other (crawling or downloading)—leaving the user to have to manually crawl the pages they want before using a downloading application or having to manually start a download of each file a crawling application finds. By combining both of these into one application, a crawling and downloading session can go unattended until it finishes, at which point the user has

quick, offline access to the media they desired. Furthermore, information can be shared between the crawler and the downloader to provide more intelligent services to the user, such as smart file naming based on the title of a web page. Secondly, its web page crawling techniques are up to par with other popular online media detectors (which typically only deal with video and/or audio). Not only that, but File Harvest's crawler also provides an advanced set of filters to use while the crawler and downloader run so that the user can download only what is of interest to them. Of course, the more specific a user desires to be in configuring a session, the more time they will have to typically spend. This can be partially avoided by remembering user preferences for a group of websites.

Current efforts to detect online media typically only address the crawling aspect of File Harvest. Even then, they are typically built into web browsers (such as Mozilla Firefox) as extensions rather than separate desktop applications. This forces users to use the specific web browser the extension supports. File Harvest uses an internal web browser to load and run the web pages it crawls. The current architecture uses Internet Explorer, but this choice is not exposed to the user of the application.

Section 3: Methodology

The primary difficulty in crawling for online media is that most websites use a combination of HTML, JavaScript and web browser plug-ins to show various types of media – typically audio and video, but sometimes also images and documents. Thus, static analysis of a web page's HTML alone is insufficient to find media on the page because the media's container is often loaded dynamically using JavaScript and then the

actual media is loaded and displayed in the media container. Thus, a component (e.g. web browser) is needed to parse and run the web page so that dynamically loaded content as well as browser plug-ins run as they normally would. However, this presents a significant challenge: How to crawl the web browser plug-in's content? To partially solve this, a proxy server is run that processes outgoing HTTP requests (and their responses) and watches for online media in the requests and responses. This approach fails to detect any online media requested through a protocol besides HTTP. Additionally, media protection schemes can ultimately thwart any attempt to automatically crawl for online media.

Various advanced downloading applications have existed for quite a while. The downloading component of File Harvest simply aims to mimic much of the behavior of these past and present applications. Such applications allow simultaneous downloads, pausing and resuming of any download, and can download each file to a customizable location on the user's computer. While many advanced downloaders actually use segmented downloading—where the download is split into many segments and each segment is downloaded and then finally merged with the other segments – File Harvest's downloader does not implement segmented downloading.

Overall, File Harvest can greatly speed up the discovery and downloading of media for its users.

Section 4: Organization of this Thesis

This thesis is divided into six chapters. In the first chapter, an introduction and overview of this thesis is given. In the second chapter, past and present work in the fields of web crawling and downloading is evaluated and compared with File Harvest. Chapter three describes the user interface for File Harvest and also discusses its purpose and goal in more abstract terms. Chapter four explains the crawling methodology of File Harvest and Chapter five describes the downloading methodology of File Harvest. Finally, chapter six touches on possible future work that could be done on File Harvest.

CHAPTER 2 - RELATED WORK

Section 1: Overview

The primary differences between File Harvest and other related software are the level of automation and the level of completeness that File Harvest offers the user. In terms of automation, File Harvest only requires a brief setup before automatically crawling and downloading media. In terms of completeness, File Harvest unifies both crawling and downloading in one single application.

These differences are explored in more detail in the following sections, where File Harvest is compared with various other applications in terms of automation and completeness.

Section 2: Related Crawling Applications

At the most basic level, we seek to review current crawling applications capable of identifying content that is dynamically loaded onto a web page through a variety of mediums. This content could be something simple like an image loaded JavaScript, or something more complex like a Flash video loaded onto a web page through the Macromedia Flash runtime.

For instance, take the YouTube video – at URL: http://www.youtube.com/watch?v=_OBlgSz8sSM : “Charlie bit my finger - again !”. A screenshot of the video is shown below in Figure 1.

Figure 1 - YouTube video we desire to find through crawling.

Needless to say, with 90 million plus views, it is a very popular video. Our goal for this test is to find a crawling or media capturing application that, given the YouTube page URL above, can identify the Flash video file URL that is being played on the page.

Two tables follow. In the first, a selection of crawling and media capturing applications, along with their associated website and their user interface method is shown. The selection is certainly not complete, but it does suffice to make a point. The second table shows how the crawling applications compare and the third table shows how the media capturing applications compare.

Listing of Crawling & Media Capturing Applications		
Application	**URL**	**User Interface**
Aaron's WebVacuum	http://www.surfwarelabs.com/	Standalone Desktop App
Fast Video Downloader	http://www.applian.com/fast-video-download/	FireFox Extension
File Harvest	NA	Standalone Desktop App
Flash & Media Capture	http://www.metaproducts.com/mp/Flash_and_Media_Capture.htm	Internet Explorer Toolbar

Flash Video Resources	http://max.subfighter.com/flv/	FireFox Extension
GetBot	http://www.getbot.com/	Standalone Desktop App
Internet Download Manager	http://www.internetdownloadmanager.com/welcome.html	Standalone & Browser Integration
Jaksta Streaming Media Recorder	http://www.jaksta.com	Standalone Desktop App
Media Converter	http://www.mediaconverter.org/	FireFox Extension
Replay Media Catcher	http://applian.com/download-videos/	Standalone Desktop App
Visual Web Spider	http://www.newprosoft.com/web-spider.htm	Standalone Desktop App

Table 1 - Listing of crawling and media capturing applications.

Comparison of Crawling Applications http://www.youtube.com/watch?v=_OBlgSz8sSM			
Application	**Can Download**	**Found YouTube, Flash Video**	**# of URLs Found on YouTube Page**
File Harvest	Yes	Yes	212
GetBot	Yes	No	206
Internet Download Manager	Yes	No	299
Visual Web Spider	Yes	No	191

Table 2 - Comparison of Crawling Applications and File Harvest

Comparison of Media Capturing Applications http://www.youtube.com/watch?v=_OBlgSz8sSM		
Application	**Can Download**	**Found YouTube, Flash Video**
Aaron's WebVacuum	Yes	No
Fast Video Downloader	Yes*	Yes
File Harvest	Yes	Yes
Flash & Media Capture	Yes	Yes
Flash Video Resources	Yes	Yes**
Jaksta Streaming Media Recorder	Yes	Yes
Media Converter	No	Yes**
Replay Media Catcher	Yes	Yes

Table 3 - Comparison of Media Capturing Applications and File Harvest

* Download is possible, but it relies on the FireFox browser download manager or other installed download managers on the user's system. There is no direct downloading through the program.

** Loads a new browser window which then does some additional work – resulting in a link/button to click on to download the video. This indirection (loading a new web page) is undesirable for the user because it takes more time and eliminates a level of automation.

In general, many other applications exist that can find the Flash video on the YouTube webpage. However, all other applications besides File Harvest have downfalls such as being tied to a particular browser—FireFox for example.

The comparison of the applications highlights the point that there are no pure crawling applications that can also find dynamically loaded content (such as Flash)—backing up the claim that File Harvest offers a new level of functionality.

Differences in the number of URLs found tend to involve trivial URLs (e-mail URLs, JavaScript URLs, and so on). It is not necessarily good to have more URLs; it is simply another measure of each crawler on this given page. Of course, finding only a small set of URLs when there should be many more is problematic. In the case of the four web crawlers, the range of URLs found is 191 – 299. This is a rather large range and is probably due to such differences as image tags or JavaScript URLs—though it is hard to say because the internal workings of the other three applications are not known.

There are some nice features to these applications. For instance, GetBot includes a “Site Explorer”, which allows the user to navigate a web site in the program using the site’s tree structure. This is useful if you do not already know what files you want to download and would rather interactively explore the website. Of course, this interactive

exploration is not the focus of File Harvest, which aims to provide a fully automated crawling and downloading experience to the user. Visual Web Spider a very nice web crawling wizard that helps users start a web crawling sessions. In another example, Replay Media Catcher and the Jaksta Streaming Media Recorder are capable of capturing media that uses a protocol besides HTTP. Among other non-HTTP protocols, they are able to capture the Real Time Messaging Protocol (RTMP) [4], which is currently being used to stream audio and video on various websites; including some very well known sites such as Hulu (http://www.hulu.com) and Yahoo! Music Videos (http://new.music.yahoo.com/videos/). Both of these features would be nice to have in future versions of File Harvest, with priority given to capturing RTMP streams.

Since there appear to be no crawling and downloading applications that can also capture dynamic media—such as Flash video—the rest of the chapter focuses on applications that can either crawl websites or that can capture dynamic media. However, it is important to remember that applications must be able to do both to truly meet the requirements we desire.

Section 3: Related Downloading Applications

Downloading applications (referred to as "download managers" or "downloaders") come in every shape and size and for all sorts of different purposes. One of their purposes is to download one or more files from the Internet to the user's computer. At the same time, many download managers act as media capturers, which can be viewed as single-level web page crawlers.

There are some common attributes of downloaders; shown in order of importance:

- Can download two or more files simultaneously.
- Integration with various web browsers expedites initiating downloads and increases ease of use.
- Settings such as customizing the download location and virus scanning files when they finish downloading.
- Claim to be able to speed up downloading using mirror locations (other web servers that have the same file, but may provide faster transfer rates) as well as segmented downloading.

A listing and comparison of download managers follows. The comparison again highlights if the downloaders are able to capture and download (assumed) the YouTube video shown earlier.

Listing of Downloading & Media Capturing Applications		
Application	**URL**	**User Interface**
Download Accelerator Plus	http://www.speedbit.com/dap/	Standalone Desktop App & Browser Plug-in
File Harvest	NA	Standalone Desktop App
FlashGet	http://www.flashget.com/index_en.htm	Standalone Desktop App & Browser Plug-in
Free Download Manager	http://www.freedownloadmanager.org	Standalone Desktop App
Fresh Download	http://www.freshdevices.com/freshdown.html	Standalone Desktop App & Browser Plug-in
GetRight	http://getright.com/	Standalone Desktop App & Browser Plug-in
Internet Download Manager	http://www.internetdownloadmanager.com	Standalone Desktop App & Browser Plug-in
Orbit Downloader	http://www.orbitdownloader.com/	Standalone Desktop App & Browser Plug-in
ReGet Deluxe	http://deluxe.reget.com/en/features.htm	Standalone Desktop App & Browser Plug-in

Table 4 - Listing of downloading and media capturing applications.

Comparison of Downloading & Media Capturing Applications		
Application	**Found Flash Video**	**Protocols Supported**
Fresh Download	No	HTTP(S)
Download Accelerator Plus	No	HTTP
File Harvest	Yes	HTTP
FlashGet	Yes	HTTP, MMS*, RTSP**
Free Download Manager	Yes	HTTP(S)
GetRight	No	HTTP(S)
Internet Download Manager	Yes	HTTP(S), MMS
Orbit Downloader	No***	HTTP(S), MMS, RTSP, RTMP
ReGet Deluxe	No	HTTP(S), MMS, RTSP

Table 5 - Comparison of downloading and media capturing applications.

*Multimedia Messaging Service

**Real Time Streaming Protocol

***Their official site says it can find YouTube video. However, I was unable to actually achieve this using the application.

The listing of applications reveals an interesting pattern. Namely, that almost every download manager has a desktop interface as well as a plug-in interface for one or more browsers. The primary reason for the plug-in interface is to have access to the web browser's requests and responses for easier web proxying as well as to enhance the program's ease of use. Also, since these applications often plug into various browsers, their primary interface is still a desktop application so that they can remain browser independent.

The comparison of applications shows that many downloaders are moving towards being able to download more dynamic media (such as Flash video). This is even

more evidence for creating a crawling and downloading application as that is already the direction many of these applications are heading in; whether they realize it or not. In fact, crawling applications generally want to help their users not only find media but also download it. The following Venn diagram demonstrates the void that File Harvest fills:

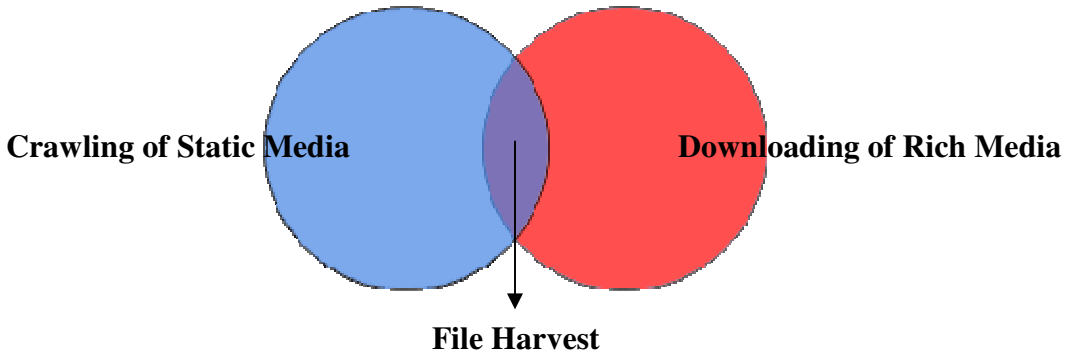

Figure 2 - The application gap that File Harvest fills.

The Venn diagram shows that File Harvest can crawl for, and download, static and rich media.

Some downloaders, such as Free Download Manager, even include a "Site Explorer" - a feature we saw in some of the crawling applications. This combination of crawling and downloading is natural; downloading applications generally want to help their users not only download media but also find it.

Protocol support is the key differentiating feature of the download managers. It essentially boils down to two groups of protocol support: those that support only HTTP and those that support HTTP, MMS, RTSP and RTMP. Many Flash video distributors use RTMP to distribute their video, so those downloaders that support it have a strong advantage against other download managers because support for RTMP and other protocols is difficult.

At the same time, supporting protocols such as MMS, RTSP and RTMP can be seen as additional and unnecessary, simply because so long as the application supports HTTP, it can capture and download the vast majority of online media. Going forward, support for RTMP would be desirable and would create a more compelling application.

CHAPTER 3 - FILE HARVEST: AN APPLICATION OVERVIEW

Section 1: Overall Approach and Goal

Perhaps, the best way to understand what File Harvest tries to accomplish is through a simple diagram.

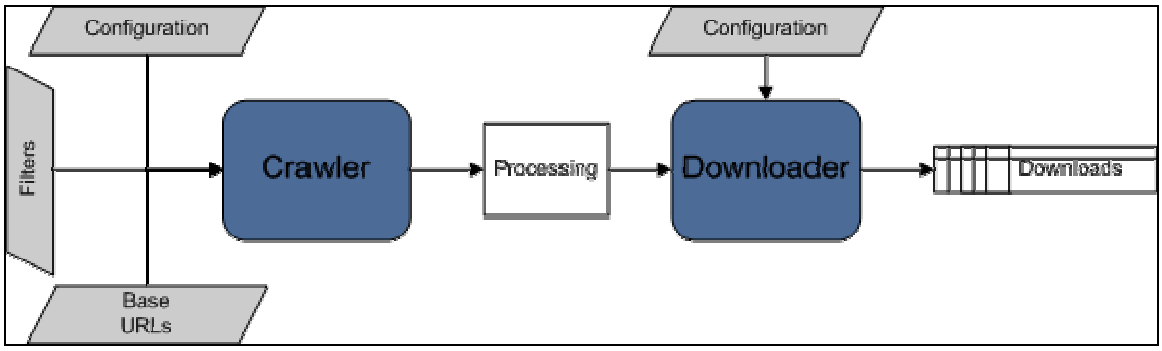

Figure 3 - File Harvest's approach and goal.

In Figure 3, the grey rectangles represent work File Harvest's user must do, the purple rectangles represent the primary work done by File Harvest (and the focus of this thesis), and the white rectangles represent work that may be done by the user and/or File Harvest.

Starting from the left of the figure, inputs go into File Harvest's crawler. The crawler does some work and reports the URLs it found to be downloaded and some information about each download. Next, additional processing may be done on the downloads. This could include filtering out other downloads or setting the download locations of some downloads. After some processing, the downloads are sent to the downloader along with some more configuration information. The downloader downloads the files the user wants to the user's computer for offline browsing.

Section 2: User Interface

The importance of an effective and easy to use user interface for File Harvest can not be understated. First of all, the configuration work required to setup a crawling and downloading session can require technical savvy – recognizing page URLs, noticing site structure patterns, and possibly constructing terms and regular expressions for filters – leading to the need to simplify these processes as much as possible. Furthermore, making such processes appear laissez-faire will encourage usage and adoption of advanced features that will give the user more power.

The user interface consists of several windows, each of which is shown below starting with the main File Harvest window.

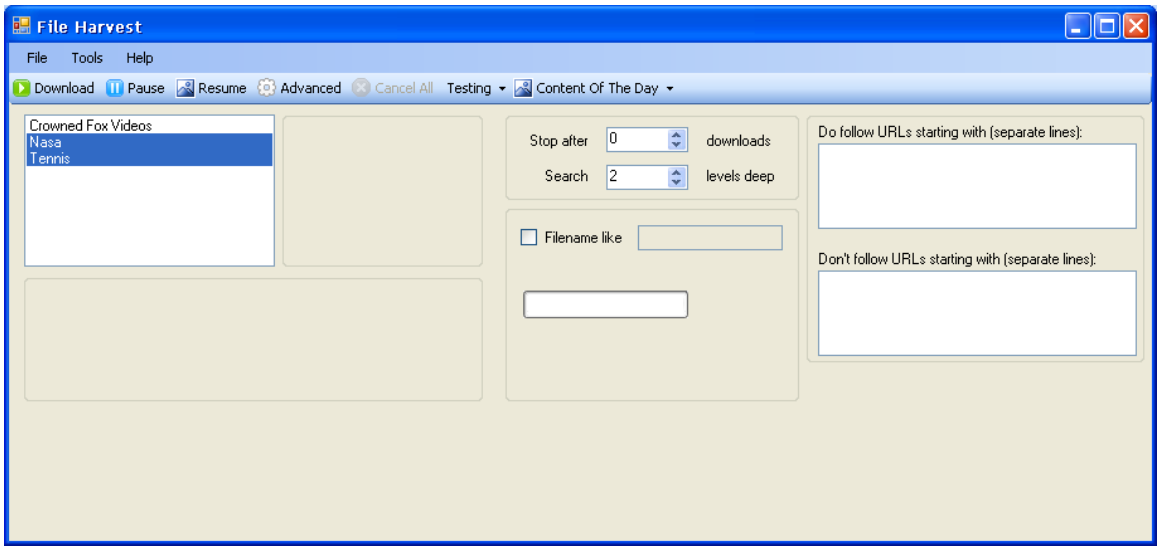

Figure 4 - The main File Harvest window.

Figure 4 shows the main File Harvest window. This window is primarily used to start, pause, resume and cancel a crawling and downloading session. To do this, the user first selects the collection(s) they want to crawl and download. Next, the user needs to change any crawling and downloading settings that they want to (e.g. crawling depth).

They can change some of these settings in the main window and others from the Options window. Finally, the user hits the "Download" button and crawling and downloading will begin. In the main window, crawling and download is not taking place and so parts of the window are blank.

The main window also contains a few key settings. For instance, the user can specify how many levels deep the crawler should search using the up/down box. The user can also list parts of URLs to use as crawling filters. In Figure 4, the user has specified that the crawler should not crawl any URLs that start "http://www.ads.com".

Lastly, the main window contains the main menu and the main toolbar. The main menu allows the user access to the many other windows of the application such as the Collection Manager. The main toolbar shows actions the user can perform in the main window such as retrieving the content of the day or starting a crawling and downloading session.

When the program is executing a crawling and downloading session, its main window shows the progress of the crawling and downloading. Figure 5 shows the main window when File Harvest is crawling.

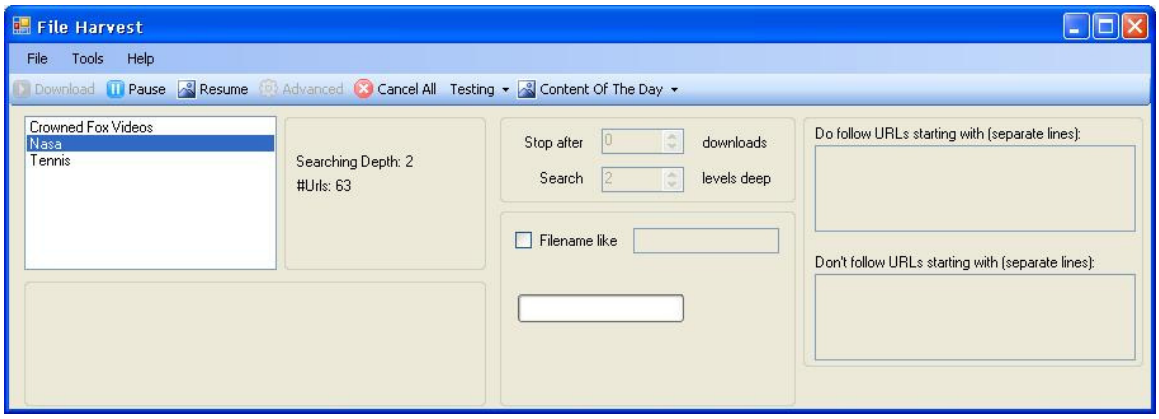

Figure 5 - The main File Harvest window during crawling.

There are not very many changes when File Harvest is crawling. First, some of the toolbar buttons have been disabled as they represent invalid actions given the crawling state of the application. More importantly, the window shows that the crawler is supposed to crawl 2 levels deep and it is currently crawling in level 2 – the last level. Also, the crawler has 63 URLs left to load and crawl for media. These 63 URLs represent a subset of the URLs that were found while crawling in level 1. It is a subset because this screenshot was taken after the crawler had already started crawling in level 2.

Once the crawler is done, downloading of the media that was found begins. Figure 6 shows what the main window looks like during downloading.

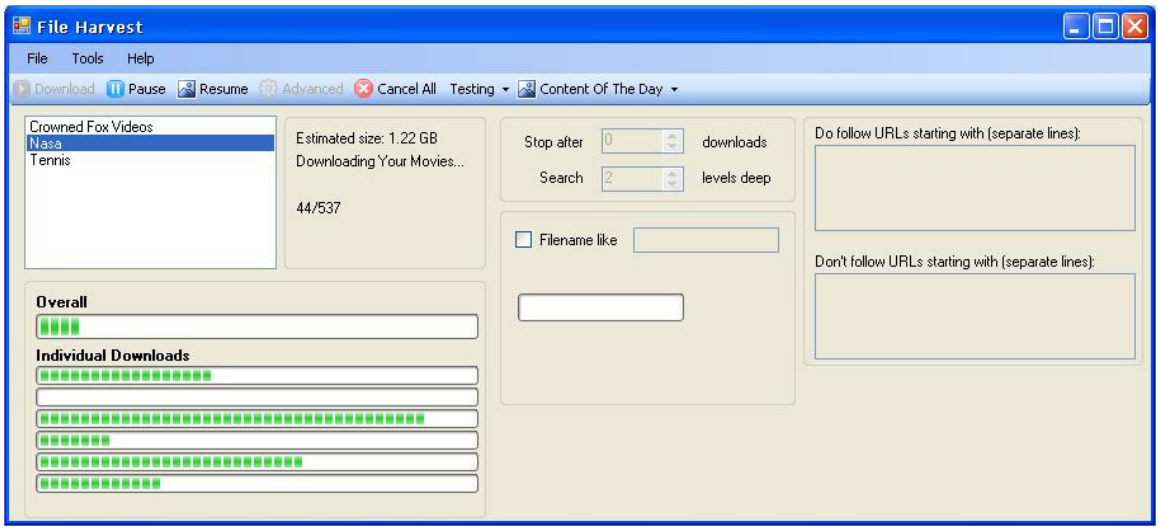

Figure 6 - The main File Harvest window during downloading.

There are a couple points of interest in the downloading view. First, the overall progress of all 537 downloads is shown in a progress bar; 44 of the 537 downloads have finished. Also, the currently downloading downloads' progress is shown. File Harvest has estimated the total size of the 537 downloads using it past download statistics—in this case, 1.22 GB.

Next, the Options window helps users set critical application settings:

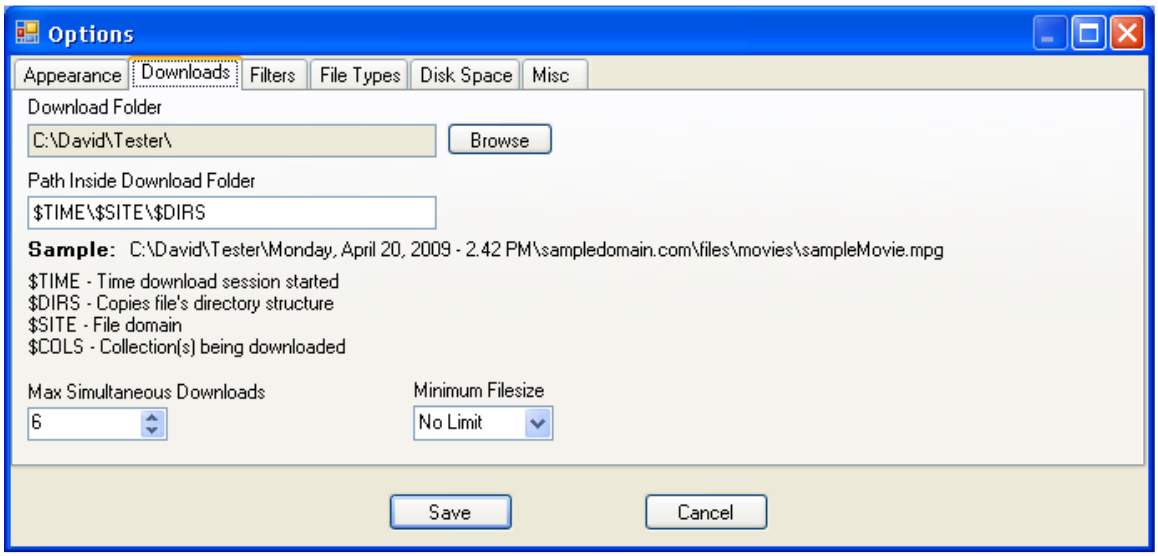

Figure 7 - The Downloads tab of the Options window.

Figure 7 shows the Downloads tab of the Options window, which also contains several other tabs of options. The Options window is where users will spend a good amount of time configuring options for their crawling and downloading. Therefore, it is also one of the areas that can undergo the most scrutiny and enhancements for the user's sake.

The Downloads tab lets the user set important downloader settings. For instance, they can choose the base download folder. This is the root folder under which all files downloaded will be saved, though the root folder may contain many subfolders that contain the downloaded files. The user can also set a dynamic file download path that changes on a file by file basis. This feature allows the user to be very specific about where their files are downloaded to. Below the box for this feature is a sample path so that the user can easily see how their changes in the dynamic path will affect the overall

path. Another useful option is that the maximum number of downloads that are occurring at once can be set. This feature is desirable for an advanced user who has good knowledge of their Internet connection and can gauge how many simultaneous downloads their computer can handle optimally. Lastly, the user can specify a minimum file size that a file must have to be downloaded. This is useful to, for example, filter out very small images versus larger, desktop wallpaper-sized images.

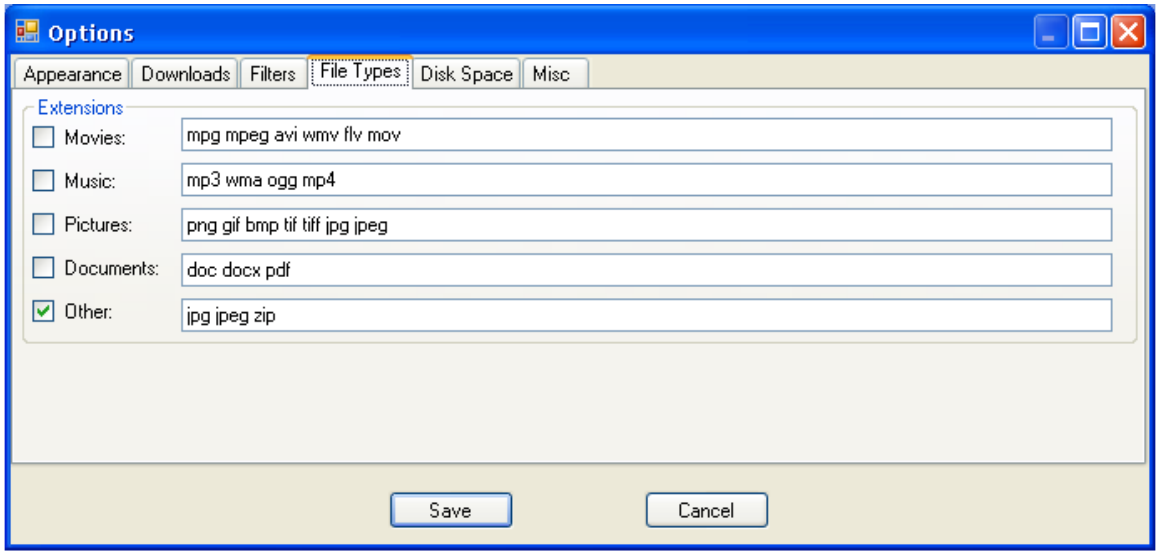

Figure 8 - The File Types tab of the Options window.

The File Types tab of the Options window is shown in Figure 8. This window's role is to allow the user to specify which file extensions the crawler should search for while exploring web pages. For instance, in the figure only files with the extensions ".jpg", ".jpeg" or ".zip" would be passed along to the downloader even if there were files with the ".doc" or ".mp3" extension on the same page. The check boxes allow the user to not have to erase extensions they may have already types. The "Other" box is useful when the extensions the user wants are not already in another grouping or when the user wants to only download one or two extensions rather than a whole grouping.

The File types tab could use a significant makeover. Although it is nicely categorized, users will often find themselves erasing and changing the "Other" box when they want to target only a few extensions.

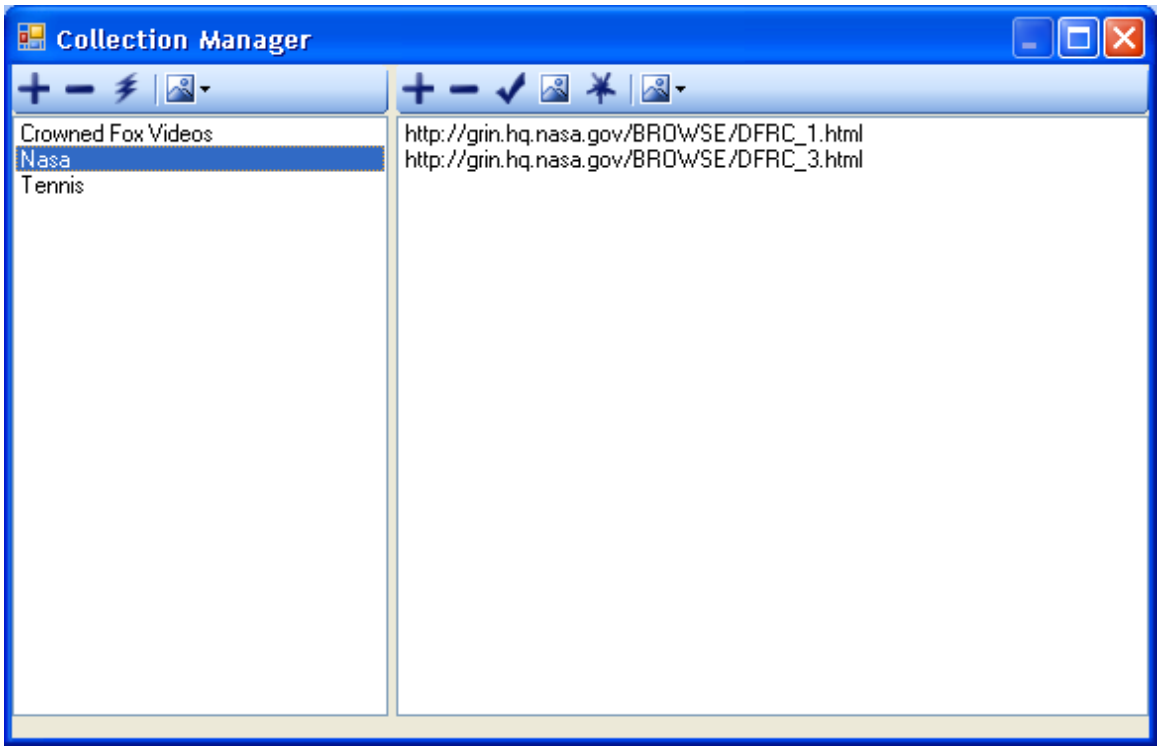

Figure 9 - The Collection Manager window.

Figure 9 shows the Collection Manager window. This window allows users to create, edit, and delete collections from their list of collections. It also allows them to explore each collection and add, edit and delete URLs from each collection.

In the figure above, there are three collections ("Crowned Fox Videos", "Nasa" and "Tennis"). Furthermore, the "Nasa" collection contains two URLs that are shown in the right hand text box.

The Collection Manager's toolbar allows the user to add (plus sign), delete (minus sign), make default (lightning bolt) and import/export (mountain) collections. Making a

collection the default collection means that when the Collection Manager window opens, that that collection is selected (even if it is not the first in the list).

The toolbar also allows the user to add (plus sign), delete (minus sign), verify (check), split (mountain), pattern generate (web) and import/export (mountain) URLs. Splitting one or more URLs means that the user can create a new collection containing the selected URLs (to be split). Verification allows the user to check one or more URLs to see if the site at the URL is reachable. This helps eliminate URLs from the list that may have gone down after a long period of time or to detect URLs that may have been incorrectly entered. Pattern generation of URLs is a very powerful tool that is shown and explained below in Figure 10.

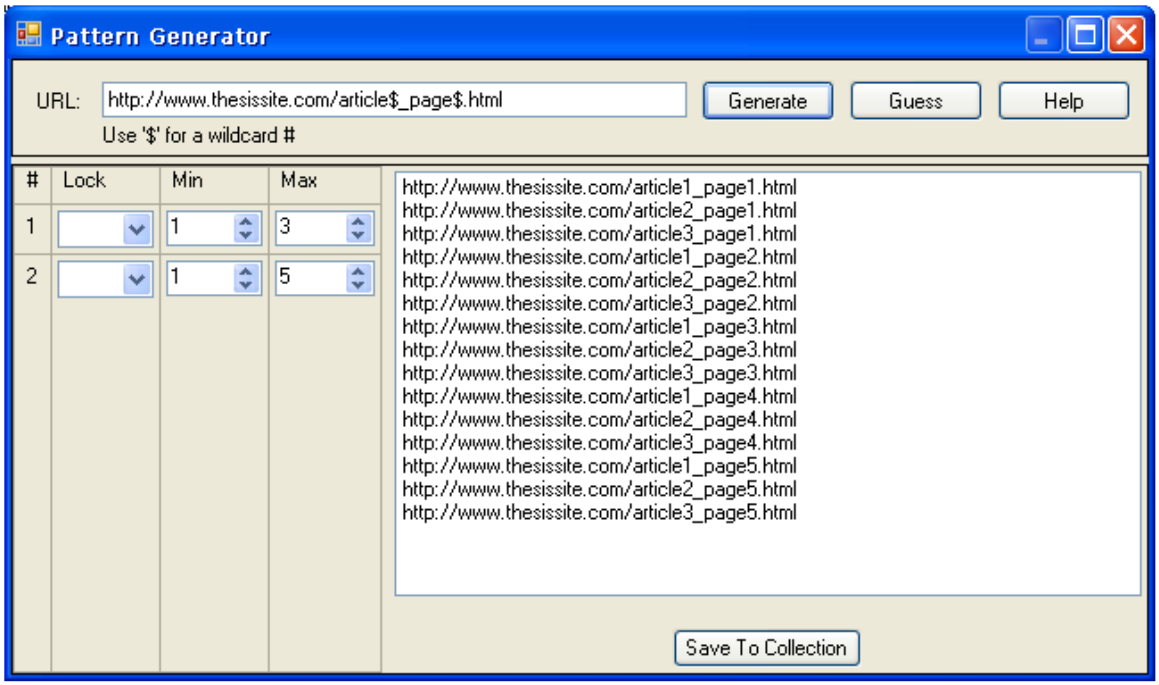

Figure 10 - The Pattern Generator window.

The Pattern Generation window, shown in Figure 10, lets a user add one or more URLs to a collection by generating URLs from a base URL. The base URL might have a

particular pattern that the user notices. For example, there may an image gallery with URLs like:

- http://www.gallery.com/pictures1.html
- http://www.gallery.com/pictures2.html
- http://www.gallery.com/pictures3.html

In this case, the base URL would be “http://www.gallery.com/pictures$.html”, where the ‘$’ represents a variable number. Then the user could set the min/max values for that variable value and then generate URLs. This feature only supports substituting integers for the variables. It would be useful to also allow letters and dates to be used.

The “Guess” feature simply replaces any integers with a ‘$’ in the URL if they occur after the domain name. So, if you pasted “http://www.gallery.com/pictures1.html” into the URL textbox and pressed “Guess”, it would become “http://www.gallery.com/pictures$.html”. Of course, you can manually do this too.

Lastly, the “Lock” feature helps in situations where the URL has two or more integers in it that are the same. For instance, take the URLs “http://www.gallery.com/day1page1” and “http://www.gallery.com/day2page2”, which seem to show a pattern where both numbers are equal. Also, pages where the integers are different (i.e. “http://www.gallery.com/day2page1”) do not exist. Thus, the user does not want to generate all URLs possible from page 1 to n (n*n), but rather just n URLs. In this case, the user would still use the base URL “http://www.gallery.com/day$page$”, but would lock either variable #1 to variable #2 or vise versa and then set both to range from 1 to 2. This would only generate n URLs as the user desires.

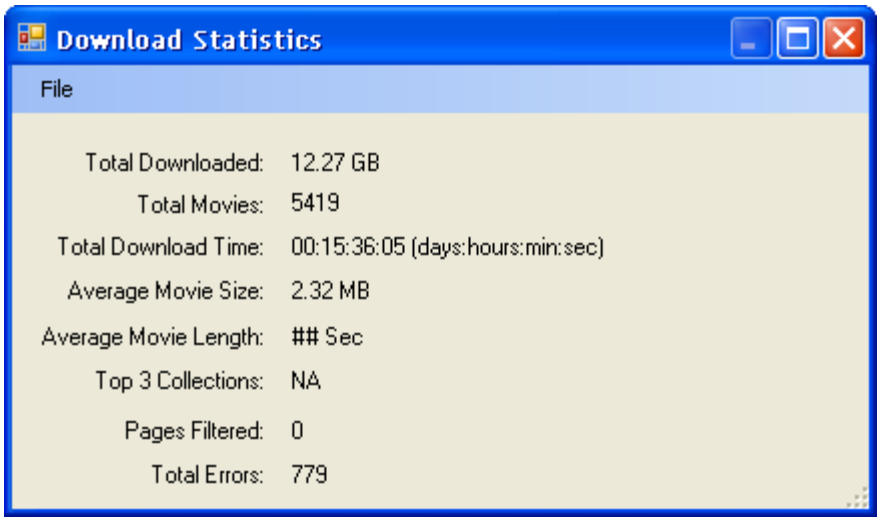

Figure 11 - The Download Statistics window.

Figure 11 shows the Download Statistics window. This window shows usage statistics for the user's crawling and downloading sessions. Although most fields say "movies", these are simply just any type of file (such as an image or document); except the "Average Movie Length" field, which attempts to actually show the average length of all (strictly) movies downloaded. However, this feature does not yet work and may ultimately be removed as it does not add enough value given its development cost.

To clarify, "Errors" could represent anything from an unreachable URL (e.g. 404 HTTP error code) to a download that is rejected because its file size is below the minimum download size allowed. Thus, even though the field is not a great source of information it is still shown for completeness.

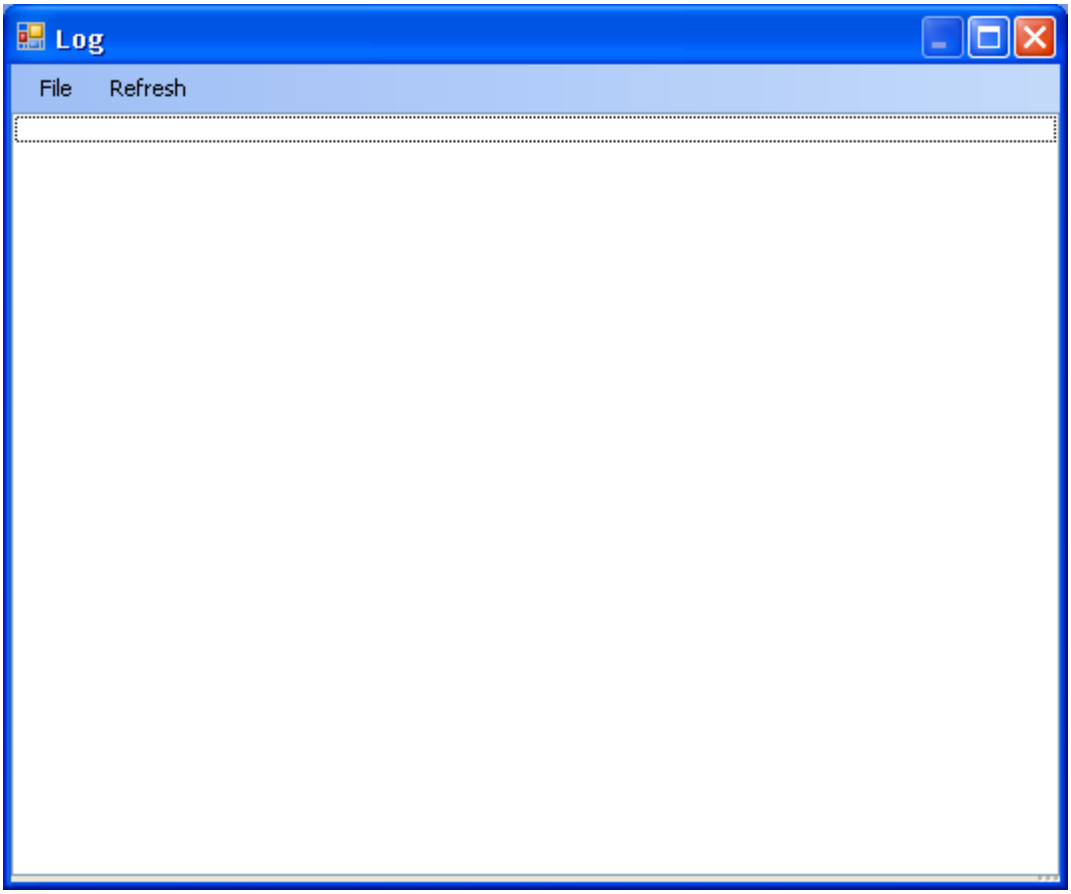

Figure 12 - The Log window.

Lastly, and most simply, is the Log window shown in Figure 12. The Log window simply shows any error messages and any important messages the user needs to see. This could be a message about a URL that was unreachable or a message noting that there was no internet connection available to being a crawling and downloading session.

The refresh button allows the user to update the Log in case any new messages have been entered into the Log since the user opened the Log window.

Section 3: Pause and Resume Support

A crawling and downloading session can be paused at any point. When a session is paused the state of the session needs to be saved so that it can be resumed. The heart

of the pausing and resuming support in File Harvest is actually supported by the pausing and resuming abilities of the crawler and downloader—as opposed to a separate pausing and resuming component.

If a session is paused during crawling, File Harvest saves the remaining URLs to crawl as well as the current URL's status. If it is paused during downloading, File Harvest saves the progress of the current downloads.

Of course, pausing a session is pointless if the session can not be resumed. If the session is resumed during crawling, the crawling component is told to resume. In doing so, it starts the crawling process over for the rest of the URLs it has yet to crawl. Resuming the downloader is more interesting. The "Range" HTTP header allows a client to request a file from a web server starting from an arbitrary point in that file. This is the means by which resume is supported not only in File Harvest, but also in any other type of download manager. However, web servers have the right to deny the use of the Range header. In this case, a file download can not be "resumed". Instead, the file download will have to be restarted from the first byte in the file. In the case of a small download, this is not a big problem. In the case of a large download though, this can greatly impact the total crawling and downloading session time. That being said, most web servers do support the Range header.

Section 4: Filtering

Much of File Harvest's power comes from its many crawling, content and downloading filters. Filters ultimately help the user only crawl for and download the

media they want – even when there is a myriad of unwanted media on the same web pages.

In general, setting up a filter adds onto the total time required to start a crawling and downloading session. This is not desirable, but the time saved in not crawling certain pages and not downloading certain media can easily offset this time and surpass it. Of course, this is time the user has to spend doing manual work. In comparison, the time associated with unwanted crawling and downloading can be taken by automatic work rather than manual user work. In any event, filters are an essential component of File Harvest and will also be a point of interest in future work.

In the following sections, the general filter structure is described and then each specific type of filter is explained.

4.1: The IFilter and IFilterSet Interfaces

To facilitate filtering for different purposes, a generic filter interface, IFilter, was defined. This C# interface is shown below:

```
interface IFilter<T>
{
        Func<T, bool> filter { get; }
        string Name { get; }
}
```

Figure 13 - The IFilter interface.

An IFilter is a parameterized type that can filter the parameterized type and return a Boolean value saying whether or not the content passed that particular filter. Additionally, the filter has a name to identify it. This way, if content does not pass a filter, the name can be used to help the user understand why that content failed.

Furthermore, in order to package many filters the IFilterSet interface was defined and is shown below:

```
public interface IFilterSet<T>
{
        void AddFilter(IFilter<T> filter);
        bool Filter(T content, out string failedFilter);
        void Clear();
}
```

Figure 14 - The IFilterSet interface.

This interface allows IFilters to be added to it and it can also be cleared. The most important function is the Filter function of the IFilterSet interface. This function actually runs each of its IFilters on the content. It can additionally report which filter a piece of content failed on – if any.

By default, if there are no filters in a filter set, then all content will pass the IFilterSet. However, if there are one or more IFilters into the IFilterSet, then content must pass those IFilters.

4.2: Crawling Filters

Crawling filters are those filters that can reduce the number of URLs to be crawled. Crawling filters and the crawling filter set implement both filter interfaces with the .NET Framework Uri class. The Uri class captures, among other things, a URL. Crawling filters are primarily concerned with checking the URL's domain, path, query string, etc…

The following table lists and describes each of the already defined crawling filters:

Listing and Description of Crawling Filters	
Filter and Parameters	**What it Filters Out**
UriMatchesFilter(IEnumerable<Regex> regexes)	URLs that do not match any of the regular expressions in regexes.
UriMustContainFilter(IEnumerable<String> uriFragments)	URLs that do not contain any of the strings in uriFragments.
UriStartsWithFilter(IEnumerable<String> startingFragments)	URLs that do not start with any of the strings in startingFragments.
UriDoesNotStartWithFilter(IEnumerable<String> startingFragments)	URLs that do start with any of the strings in startingFragments.
UriDomainIsFilter(IEnumerable<Uri> domains)	URLs whose domain is not any of the domains in domains.
UriDomainIsNotFilter(IEnumerable<Uri> domains)	URLs whose domain is any of the domains in domains.

Table 6 - Listing and description of File Harvest crawling filters

The crawling filters are useful for not crawling URLs outside of the base URL domains, or not crawling URLs that contains words like “ads” in them, which will probably not yield any interesting content to the user. Also, the crawling filters can help to very specifically trace a path through a site structure, perhaps only downloading certain images for a large image gallery online.

4.3: Content Filters

Content filters allow the user to filter out web pages based on the content of the web page. The following table lists and describes the only current content filter:

Listing and Description of Content Filters	
Filter and Parameters	**What it Filters Out**
BannedWordsFilter(IEnumerable<String> bannedWords)	URLs whose content contains any of the strings in bannedWords.

Table 7 - Listing and description of File Harvest content filters

This filter is could be used to not crawl web pages about “gambling” or about “homework”, for instance. More content filters would be desirable, but creating them

possibly means better understanding the actual content of the web page – a challenging problem.

4.4: Downloading Filters

Download filters help filter out downloads that, much like in the crawling filters, do not meet certain download URL requirements. Additionally, downloads can be filtered on other download information such as the download's MIME type or the title of the web page that the download was found on.

The following table lists and describes each of the already defined download filters:

Listing and Description of Download Filters	
Filter and Parameters	**What it Filters Out**
UriMatchesFilter(IEnumerable<Regex> regexes)	URLs that do not match any of the regular expressions in regexes.
UriMustContainFilter(IEnumerable<String> uriFragments)	URLs that do not contain any of the strings in uriFragments.
UriStartsWithFilter(IEnumerable<String> startingFragments)	URLs that do not start with any of the strings in startingFragments.
UriDoesNotStartWithFilter(IEnumerable<String> startingFragments)	URLs that do start with any of the strings in startingFragments.
UriDomainIsFilter(IEnumerable<Uri> domains)	URLs whose domain is not any of the domains in domains.
UriDomainIsNotFilter(IEnumerable<Uri> domains)	URLs whose domain is any of the domains in domains.

Table 8 - Listing and description of File Harvest download filters.

Download filters are very useful to only download media with certain MIME types or certain file extensions.

Section 5: Understanding Crawling & Downloading Through Graphs

It is useful to think of the crawling process as exploring a directed graph of nodes and edges, where the nodes are web pages and the edges are connections (usually hyperlinks) between those web pages. An edge originates from the web page with the hyperlink and points to the web page that the hyperlink references. We will represent these web page nodes using circles.

Furthermore, we will represent a download using a triangular node. The idea of a download is completely dependent on the user's configuration of the application settings and filters. In fact, every file (web page, video, picture, etc…) is considered downloadable until a user configures the settings otherwise.

Figure 15 shows such a directed graph. Nodes inside of the large circle are crawled for more links (edges) to other nodes. Additionally, all triangular nodes are downloaded. Notice how files can be both crawled and downloaded – all depending on a user's settings.

The graph is certainly not acyclic; web pages can easily (and often do) contain circular paths of hyperlinks. However, File Harvest keeps a history of what web pages it visits and will not crawl a page it has previously crawled – eliminating the potential to download the same file more than once and, more importantly, not doing repetitive work that consumes the user's bandwidth and time.

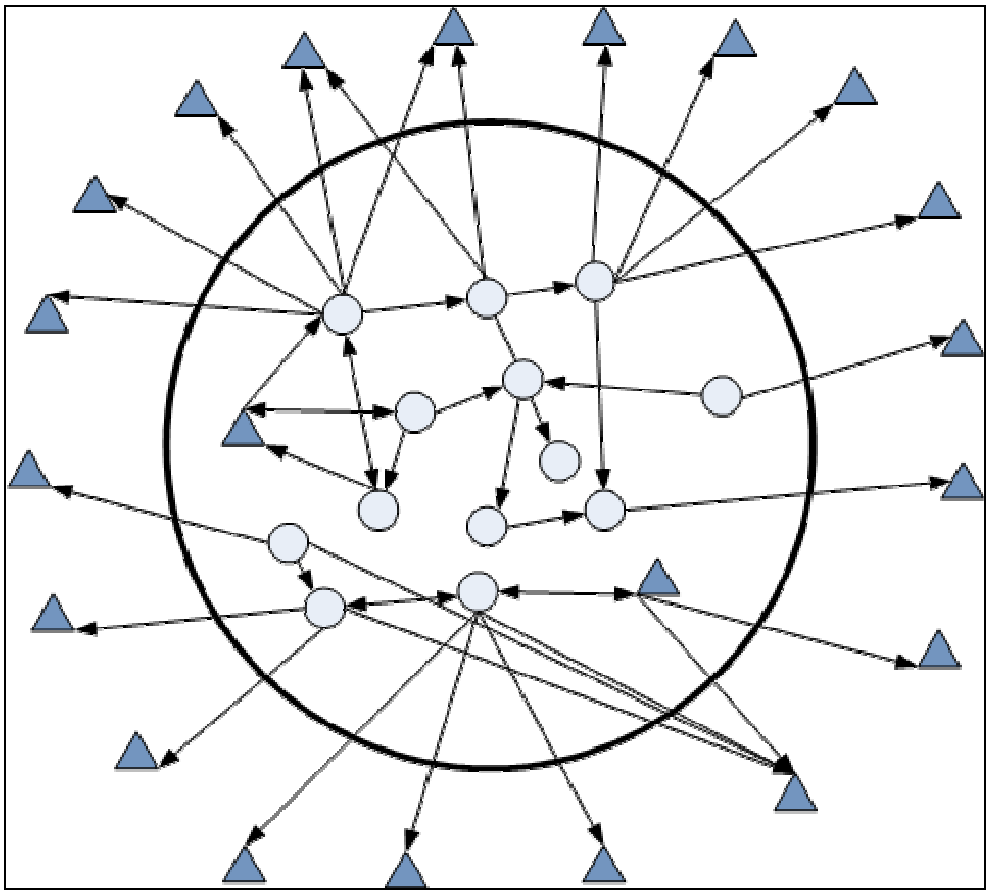

Figure 15 - A directed graph representing the idea of crawling and downloading.

Everything within the large circle is crawled – including the triangular nodes within it. All triangles represent files that were downloaded.

Figure 15 does, however, neglect two important concepts – that of base URLs and crawling depth. Base URLs serve as starting points for the crawler and the crawling depth indicates how many levels of hyperlinks the crawler will explore before it stops.

Figure 16 accurately depicts what File Harvest's crawler actually does crawl. In the figure, the red, enlarged circles represent the base URLs. Furthermore, assume the crawling depth has been set to 2. Compare Figure 15 with Figure 16 and notice how in Figure 16 there are no nodes that are more than 2 directed edges away from a base node.

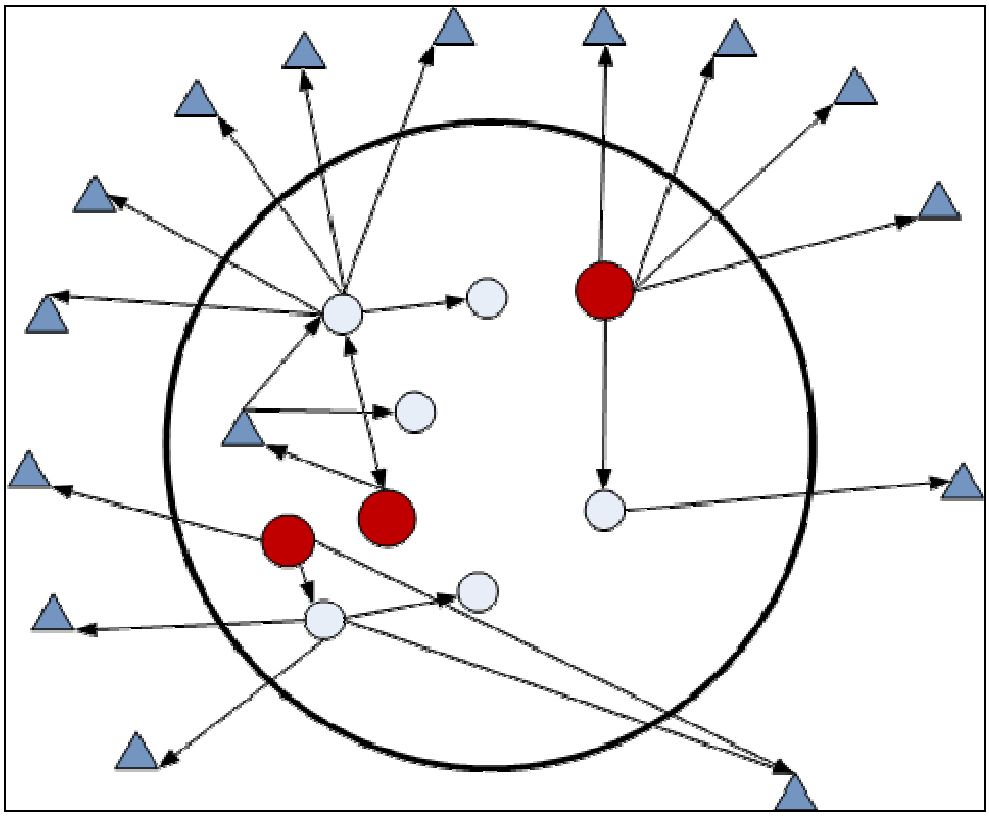

Figure 16 - Directed graph considering base URLs and crawling depth.

Also, notice how some edges have become one-way edges instead of two-way edges like in Figure 15. This also accurately captures File Harvest's behavior. File Harvest will not crawl any pages found on a page at the last level of depth.

So, how does File Harvest actually explore and crawl web pages and how does it analyze those web pages in order to find media? The next chapter answers these very questions.

CHAPTER 4 - FILE HARVEST: CRAWLING

Section 1: Identifying What Needs To Be Discovered

There are many ways in which media can be served, or delivered, to a user on a web page. One way to classify these methods is into two categories: static delivery and dynamic delivery. Furthermore, within these categories there are the specific mechanisms that are used to serve the media. Figure 17 shows the breakdown of media delivery mechanisms:

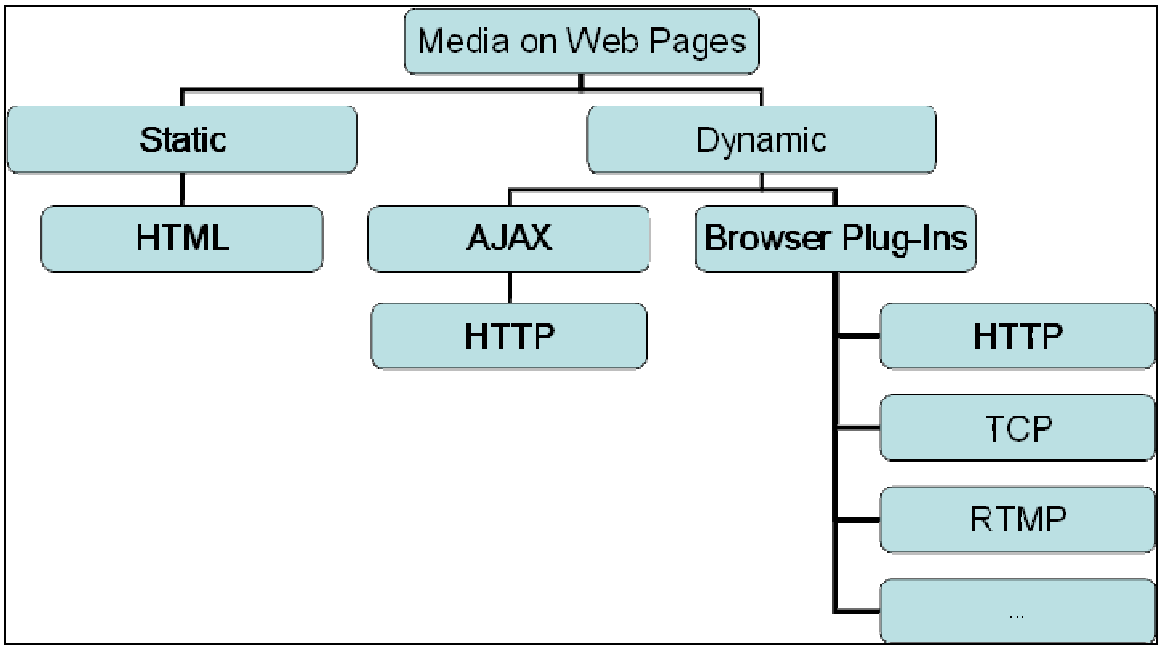

Figure 17 - Media delivery categories and mechanisms.

These categories and mechanisms will be discussed in the upcoming sections of this chapter. At this time though, it is important to know that the diagram nodes that are bolded represent categories or mechanisms that File Harvest can capture media from. Ultimately, File Harvest aims to be able to capture media regardless of how it is served and in this case the top diagram node would be bolded.

1.1: Static HTML Content

Today's Internet user encounters a multitude of media delivered in a multitude of ways. Most media is still delivered directly via standard and common HTML tags such as the image (<img …>) tag and as anchor hyperlink (<a …>) tag.

Identifying image tags and anchor tags in an HTML document is accomplished relatively simply. One method would be to use the web browser component provided by the Microsoft .NET framework to load the webpage. Then, you can access all of the images on the web page through the web browser component. Another way is to read the HTML contents of the webpage and then use regular expressions to parse the page for image and anchor tags. File Harvest uses the latter approach, which we term static HTML analysis because dynamically inserted content (through Flash or through dynamic HTTP requests via JavaScript in the HTML document) will not be captured.

1.2: Dynamic HTML Content

Web 2.0 has seen a dramatic rise in the use of AJAX, or Asynchronous JavaScript and XML. Using AJAX, web developers can dynamically load HTML content on to a web page without having to refresh the page, which has made it a very popular web development tool.

Content loaded through AJAX is harder to discover than traditional, static HTML content. This is because the dynamically loaded content is not part of the static HTML document. Thus, statically searching through a page's HTML will not reveal content loaded using AJAX. The following JavaScript and HTML code in Figure 18, obtained from [5], is an example of using AJAX where static analysis would fail:

```
//Gets the browser specific XmlHttpRequest Object
function getXmlHttpRequestObject() {
	if (window.XMLHttpRequest) {
		return new XMLHttpRequest();
	} else if(window.ActiveXObject) {
		return new ActiveXObject("Microsoft.XMLHTTP");
	} else {
		alert("Your browser doesn't support the XmlHttpRequest object");
	}
}

//Get our browser specific XmlHttpRequest object.
var receiveReq = getXmlHttpRequestObject();

//Initiate the asyncronous request.
function sayHello() {
	if (receiveReq.readyState == 4 || receiveReq.readyState == 0) {
		receiveReq.open("GET", 'SayHello.txt', true);
		receiveReq.onreadystatechange = handleSayHello;
		receiveReq.send(null);
	}
}


//Called every time our XmlHttpRequest objects state changes.
function handleSayHello() {
	if (receiveReq.readyState == 4) {
		document.getElementById('span_result').innerHTML = receiveReq.responseText;
	}
}
.... Same File
<!-- Clicking this link initiates the asyncronous request -->
<a href="javascript:sayHello();">Say Hello</a><br />
<!-- used to display the results of the asyncronous request -->
<span id="span_result"></span>

.... SayHello.txt (a different file):
Visit <a href="http://www.osu.edu">Ohio State University</a>.
```

Figure 18 - Example AJAX Code

In this AJAX code, the contents of SayHello.txt are dynamically requested. Once the contents have been loaded, they are written to the “span_result” tag. Static analysis would not find the link http://www.osu.edu that is in the SayHello.txt file because it is not in the static HTML of the page using AJAX. At most, static analysis of the page might identify the “SayHello.txt” file that is requested in the AJAX code—though this is uncommon since trying to capture such relative URLs in JavaScript is

unreliable; the value “SayHello.txt” could come from another AJAX request, from a JavaScript variable or other sources in the code.

A different type of analysis is needed to discover dynamically loaded HTML content. Since the content will be loaded using the HTTP protocol, one method would be to set up a web proxy between the web browser and the Internet. This way, when a dynamic request is made for content, through the web browser, the proxy can observe the requested URL as well as the response and report those URLs to the crawler. Figure 19 shows the File Harvest proxy sitting between the web browser (Internet Explorer) and the Internet (Web Server). Furthermore, the arrows indicate that data flows between each component; going from left to right on a request and from right to left in the response to that request.

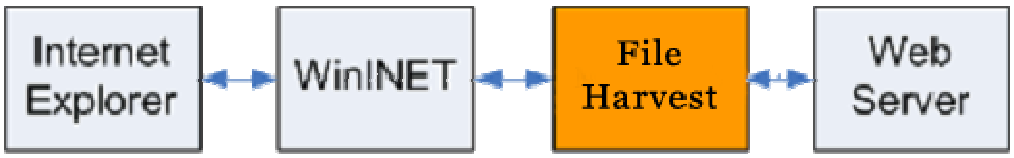

Figure 19 - File Harvest proxying WinINET.

Additionally, Figure 19 shows the WinINET component. This component is a built-in part of the Windows OS. It is actually the component that makes HTTP requests and is used by Internet Explorer (both the standard web browser as well as the .NET framework web browser component). In fact, File Harvest proxies WinINET directly and the web browser indirectly through WinINET. This could prove to be advantageous since many other applications in Windows use WinINET for HTTP calls – leading to possibilities of proxying other such applications without too much additional development work.

1.3: Content in Browser-Hosted Components

An increasingly popular way to embed media on a web page is through browser-hosted applications. These are essentially plug-ins for a web-browser. They typically run with very little restriction on the user's computer. For instance, Internet Explorer's ActiveX controls (Internet Explorer's plug-ins) run without any sandboxing [6]. Browser-hosted components are used to have more rich content on web pages, deploy specialized content, protect content via proprietary technologies, and many other purposes. One such application is Adobe Flash (Flash) and it will be the focus of the discussion of browser-hosted components going forward. A web developer can embed Flash on a web page using the standard HTML object (<object …>) and embed (<embed …>) tags [7]. Flash acts as a blank canvas for yet more content and media. Commonly, however, it is used to embed video and audio on web pages since these capabilities are not natively supported by the HTML 4 standard [8].

The difference between Flash and HTML, which is applicable to File Harvest, is the fact that Flash's content and media are not as easily obtained as media embedded in HTML. There are two main reasons for this. First, Flash files are stored in a binary format, which makes them very difficult to interpret into a meaningful set of information. Secondly, the content and media that is loaded into a Flash document can be loaded in a variety of ways. For instance, Flash may use the browser that is hosting it to load the content (e.g. Internet Explorer). In this case, the browser will be making an HTTP request on behalf of Flash. Alternatively, Flash has its own mechanisms for making HTTP (and other protocol) requests. Since there is no way to easily tell what methods

Flash is using to load media, multiple strategies need to be used to crawl for media within Flash. In both cases, there is media in Flash that we want to discover, but we will have to accommodate different media delivery techniques.

Even more disconcerting is that it is equally hard to find out if there is even relevant media in Flash in the first place. Put another way, there is no sure-fire way to tell if a Flash container is showing a video, displaying a slideshow of pictures or just has a site's navigation menu in it. Thus, File Harvest will have to either rely on the user to tell it whether or not to search for media within Flash, or it will have to assume there could be Flash on a page and therefore, by default, crawl for media embedded in Flash. What most related software does to solve this and to download the media in the first place is to hardcode mappings between certain websites and the type and location of the media on that site. For instance, the software would have a configuration for YouTube, a configuration for CNN Video and so on. Although this does ultimately allow the user to get the media they want, there is no site crawling feature and this technique does not scale nicely (only linearly as a new site comes online). Furthermore, if a site makes a change that impacts the configuration of that site within the program, users can find themselves with a broken piece of software and developers can find themselves racing to update their software to keep up with the changes on the web site.

Currently, File Harvest just captures HTTP traffic from embedded components (e.g. Flash). Figure 20 shows how File Harvest tries to detect media being loaded by Flash. In fact, notice how similar Figure 20 is to Figure 19.

Figure 20 - Flash HTTP requests being proxied by File Harvest via WinINET.

This means that File Harvest is capable of capturing some, but not all, media embedded in Flash. For instance, Flash can use alternative protocols to HTTP to request media. In these cases, File Harvest can not intercept the request because File Harvest does not monitor such protocols.

Section 2: MIME Types

An integral part of web crawling is identifying the type of content at any particular URL. To this extent, the Multipurpose Internet Mail Extensions, or MIME, have been developed and standardized for common content types. The MIME standard allows web servers to tell requesting clients (usually web browsers) the type of media at any particular URL that is requested. For instance, a standard web page might have a MIME type of: text/html. A text document might have a MIME type of: text/plain. As a last example, an MP3 file might have a MIME type of: audio/mpeg. In fact, the MIME standard defines classes of media as shown in Table 9:

MIME Class	Description
application	Typically, binary files. Examples include EXE and DOC files.
audio	Audio and sound files. Examples include MP3 and WAV files.
image	Image and picture files. Examples include JPEG and BMP files.
text	Text files. Typically, human readable. Examples include HTML and TXT files.
video	Video files. Examples include AVI, WMV, QT, and FLV files.

Table 9 - Listing of MIME classes with a general description of each.

Furthermore, fully qualified MIME types take the form of: <MIME class>/<encoding or file type>. While there are a few other MIME classes, the above five classes are predominate and are the classes relevant to File Harvest.

So, why does File Harvest need to know about MIME types? MIME types provide a reliable and standard way to identify the type of content located at a particular URL. Thus, this enables File Harvest to map a URL to a type of media. More specifically, File Harvest needs to know the type of media at a URL so that it knows if it can crawl it and if it should download it for the user.

File Harvest only considers crawling URLs that are in the "text" MIME class since these are easily read and parsed. The following table, Table 10, shows the MIME encoding/type that File Harvest will crawl within the text class:

MIME Text Type	Comments
html	HTML documents. This is the most common type crawled.
csv	Comma separated values.
plain	Plain text files.
richtext	More richly formatted text files.
tab-separated-values	Tab separated values.
x-setext	Lightweight markup language.
x-sgml	Standard Generalized Markup Language. Similar to XML.
xml	XML files.

Table 10 - Listing of text MIME types that File Harvest crawls.

2.1: File Extensions

File extensions are much more common to users than MIME types. However, they are not as reliable as MIME types when it comes to classifying the media type of a file or URL. This presents a problem from a user interaction standpoint since the program uses MIME types internally but needs to allow users to input file extensions since that is what users are familiar with. The mapping between MIME types and file

extensions is not 1-to-1. Instead, there can be many file extensions mapped to just 1 MIME type.

File Harvest uses a multi-set structure to store the mappings between MIME types and file extensions. This MIME-Extension multi-set supports the following operations shown in Table 11:

Function	Description
public static bool ExtensionMatchesMime(string ext, string mime)	Determines whether the extension maps to the MIME type
public static List<string> GetExtensionsForMime(string mime)	Returns a list of extensions that map to the MIME
public static List<string> GetMimesForExtension(string ext)	Returns a list of MIMEs that are associated with the extension.

Table 11 - Listing of MIME & File Extension Multi-Set class operations.

Using these functions, File Harvest can easily allow users to input file extensions in the user interface while still maintaining robust media type identification in the crawling component.

Section 3: Static HTML Analysis

Using regular expressions, File Harvest can search for media within the HTML of a web page. Although this methodology does not yield dynamically served media, it does have the advantage of yielding most media as well as being able to capture media that is not in locations that media is traditionally found. For instance, there may be a URL in the text of a web page that is not an HTML anchor (<a…>…). Figure 21 demonstrates the difference:

```
<html><body>

This page is about <a href="http://www.archive.org">links</a> and HTML.
Sometimes links like http://www.w3.org are just embedded in the text of
an HTML document.

</body></html>
```

Figure 21 - Example HTML document.

In the figure, one link is an anchor tag, but the other link is just in the text of the document without any surrounding tags.

File Harvest is able to capture both URLs in the figure if the user chooses to have File Harvest do so. Though these cases may be uncommon, they can also be useful when URLs are listed in other parts of an HTML document such as in JavaScript code. Figure 22 shows such an example.

```
<html><head>
<script language="JavaScript">
var url = "http://www.archive.org";
window.open(url);
</script>

</head><body>
A page about URLs in JavaScript.
</body></html>
```

Figure 22 - HTML document showing a URL in JavaScript code.

Again, File Harvest would be able to find these URLs because it simply searches the HTML of a web page rather than looking at a tree of page elements such as anchor and image tags. Although one could find such URLs by searching all page elements in a tree, the development time associated with building such a tree do not warrant its uses.

File Harvest uses three regular expressions to capture different amounts of media from different HTML elements. Table 12 describes each regular expression:

.NET Regular Expressions Used in Static HTML Analysis	
Regular Expression	**Description**
<a[^>]+?href\s*=[^"']*?["'](?<url>[^" >]+?)["'][^>]*?>	Anchors: Only captures URLs contained within HTML anchor tags.
<(a[^>]+?href|img[^>]+?src)\s*=[^"']*?["'](?<url>[^" >]+?)["'][^>]*?>	Anchors & Images: Only captures URLs contained within HTML anchor or image tags.
(?<url>(http|https)\://[a-zA-z0-9\-\.]+\.[a-zA-Z]{2,3}(/[a-zA-Z0-9\._,\-~\(\)]+)*/?(\?[^"' \r\n>]*)?)|(<(a[^>]+?href|img[^>]+?src)\s*=[^" "']*?["'](?<url>[^" >]+?)["'][^>]*?>)	Anchors, Images and freestanding URLs: Attempts to capture all URLs within an HTML document.

Table 12 - Listing and description of .NET regular expressions used to find URLs in an HTML document.

Although the regular expressions are long and complicated, they have simple purposes. In fact, the last regular expression is half the second regular expression and half original. The original part attempts to capture what we term “freestanding URLs”, or URLs that are not located within an anchor or image tag, but instead are located with the text of an HTML document.

Section 4: Protocol Monitoring for Dynamically Served Media

One major contribution of this thesis is the ability to capture some dynamically served media. To this end, File Harvest monitors, or proxies, protocols of interest in order to watch for media going across those protocols. Currently, File Harvest only supports the HTTP protocol (and only the GET and POST methods) and does so by proxying the WinINET component of Windows. While proxying WinINET, File Harvest invokes the .NET WebBrowser component to load and run a web page. When the WebBrowser loads and runs the web page, it makes one or more HTTP requests via

WinINET. In doing so, File Harvest actively proxies WinINET and intercepts the WebBrowser's requests. File Harvest then observes the browser request, actually makes the request on behalf of WinINET, observes the response and then sends the response to WinINET – completing the proxying steps.

Of course, this process typically occurs more than once for each page loaded into a WebBrowser. In general, a web page has links to external resources such as Cascading Style Sheet (CSS) files, JavaScript files (.js), images and more. Each request for an external resource begins the proxying process and enables File Harvest to observe the resources loaded into each web page.

This effectively means that anything the WebBrowser requests via HTTP, File Harvest can capture for download later. So, if there is a dynamic request - via AJAX technologies – then File Harvest will see that request. Furthermore, if there is a request by a hosted-browser component – like Flash – then File Harvest will also see that request. Finally, this means that there could be an AJAX request to load a Flash file and that that Flash file loads media; File Harvest would see all of those requests!

4.1: Brief Overview of HTTP Request and Response structure

To better understand what type of data and the structure of that data that File Harvest deals with while proxying WinINET it is necessary to quickly summarize the structure of an HTTP request and response. The following figures, Figure 23 and Figure 24, show a simplified, but adequate view of the HTTP request and response:

<HTTP method> <path relative to host> HTTP/<version>
Host: <host URL>CRLF
(zero or more of <example-name:example-value>CRLF)
CRLF
<request body, if POST method>

Figure 23 - General structure of an HTTP request.

HTTP/<version> <response code> <response string>CRLF
(one or more of <example-name:example-value>CRLF)
CRLF
<response body>

Figure 24 - General structure of an HTTP response.

In both figures, there is an HTTP header line specifying key settings. Additionally, both figures contain one or more header lines specifying additional properties of the request or response. After the headers, there is a Carriage-Return Line Feed (CRLF) – typed as "\r\n" – indicating the end of the headers. Lastly, there is the body of the request or response. The request typically only has a body if it is a POST request, which involves sending request data to the web site to, for instance, submit data from a form on a web page.

The simple structure of HTTP requests and responses, plus their easy-to-handle ASCII encoding, make their parsing very simple; especially given the power of the requests and responses. Given that this research is not geared toward protocol processing (among other topics), it is very fortunate that HTTP is relatively easy to work with.

4.2: Algorithm for Proxying WinINET

The algorithm used to proxy WinINET is shown in Figure 25:

1. Set WinINET proxy to port: localhost:<port>
2. Establish a TCP Listener at localhost:<port>
3. Generate HTTP traffic using WebBrowser
4. Wait for WinINET proxy to receive request from WinINET
    a. Parse the request for interesting data such as the request URL
    b. Make actual request using HttpWebRequest object.
    c. Get actual response, HttpWebResponse, object
    d. Process the response for interesting data such as responding URL, MIME content-type, etc…
    e. Package the response into a byte array to send back to WinINET.
5. Send actual response back to WinINET
6. If proxying is canceled
    a. Stop TCP Listener, Finished
    b. otherwise, go to step 4.

Figure 25 - Algorithm for proxying HTTP communication.

Step 1 of the algorithm is done programmatically through the WinINET API. Step 2 is accomplished using the TcpListener component found in the .NET Framework. Step 3 is accomplished by navigating to a web page using the .NET Framework WebBrowser.

Step 4a is accomplished by converting the byte array request from WinINET into ASCII text, checking for the HTTP protocol (as the WebBrowser may generate non-HTTP traffic), and then reading the header lines of the HTTP request for interesting data. This interesting data usually includes the requested URL, the HTTP method (GET or POST) and any connection settings. File Harvest can handle both GET and POST HTTP commands – but not any other standard HTTP commands. To implement step 4b, File Harvest creates an HttpRequest component (from the .NET Framework) and sends the request to the web server. Step 4c includes checking for many error conditions that could have occurred in the HTTP request such as various HTTP codes. For instance, the .NET Framework HttpRequest throws a WebException when the HTTP response code is Not Modified (304). This creates complexity because many different HTTP response codes

have to be handled (or simply ignored if they are not expected often enough to warrant the development time and cost). File Harvest also tries to standardize the response into objects it can work with regardless of if an exception was thrown or not. After the response has been obtained, interested data is again gathered from the response in step 4d. The data in the response is often more valuable than in the request, especially because the response contains the MIME content-type header that specifies the type of media being sent in the response. Finally, a response is built (almost entirely from the actual response) to be sent back to WinINET in step 4e. This involves sending some additional header values related to the proxying of WinINET as well as sometimes sending an empty response body – a crucial design decision.

Step 5 simply involves writing the complete HTTP response to WinINET. This effectively ends the proxying process. Lastly, the proxying component checks to see if it has been stopped or canceled yet. If so, it closes the TCP Listener (the proxy connection accepter) in step 6a. Otherwise, the TCP listener remains active on the proxy port and waits for more connections from WinINET.

4.3: Short-Circuiting the HTTP Response

As previously mentioned, File Harvest selectively sends empty response bodies back to WinINET during the proxying process. File Harvest does nothing extra besides setting the response body to an empty byte array. For instance, it does not set the HTTP response code to a different value. In doing so, it stops (or short-circuits) the reading and downloading of the response body. This is done when the following condition is met:

- The MIME content-type header of the response is a type which is believed to not generate any more requests from the web page being loaded.

This condition implies that continuing to download the response body adds no value to the user and does not cause invalid behavior in the program. Since the user never sees the web pages that are being loaded, there is no point in downloading any elements that will not generate future requests, which could possibly lead to discovering more media the user wants to download.

This is intuitively correct and will use less bandwidth and time, but it does beg the question of whether or not it is appropriate. This question is hard to answer because developers could potentially check for the completeness of a file they requested before performing other actions that would generate further HTTP traffic; even if this file is a type of media like an image or video. In general, such a check for completeness is hard to imagine being implemented and it would be a very tedious task to only short-circuit responses from certain web sites (for instance). This question is not directly answered in this thesis, but we believe that it is a safe and effective way to cut down on proxying time while still finding the media the user desires.

Furthermore, this is also a good idea when the MIME content-type header of the response is recognized as a MIME type that the user desires to download. In this case, if the proxy were to download the response body, the program would be downloading the media the user desires twice for no good reason since the media downloaded by the proxy is discarded once the proxying process is completed for a web page. Currently, there is no user setting to disable this short-circuiting; though such a feature may be implemented later if there are enough requests for it.

CHAPTER 5 - FILE HARVEST: DOWNLOADING

Section 1: Overview

File Harvest's downloader component is used to download all of the files that File Harvest's crawling component finds in a crawling and downloading session. The downloader, much like the crawler, needs to be able to run autonomously until it is finished. The downloader acts as a file copier – reading a file from a web server and then writing the same file to a local copy on the user's computer for offline browsing.

The downloader is a simple component when compared to the crawler. This stems primarily from the fact that downloading is a more common task in software and thus, proven solutions often already exist in the .NET Framework to expedite development. Still though, the downloader is the second most complex part of File Harvest only to the crawler. Furthermore, the downloader must work properly or the crawler is doing useless work as the media will never be downloaded (or downloaded improperly) to the user's computer.

Section 2: Why Build A Custom Downloader?

As mentioned in the Related Work section of this thesis, building a custom downloader may seem frivolous – there are plenty of adequate and more professionally tested downloading components available to users and developers – but there are a few key benefits. The main benefits of creating the custom downloader are: better control over filtering and better control over downloading options.

File Harvest's downloader can take various filtering actions early in the downloading process since it has direct access to the HTTP request and response. For instance, it can check the size of the download before it actually starts the download. In doing so, it can abort the download early if the file size does not meet user download size filters. This means the user will not have to wait as long for all of the media they desire to finish downloading.

Additionally, the downloader's options can be completely controlled. For instance, the downloader can be paused and resumed easily. Also, it is easier to dynamically set the download's local path. Of course, it would be easy to specify an initial local path, but what if (somehow) a file with the same name and path is created while a download (with the same final destination) is waiting in the download queue. When that download does finally start, the downloader (not the File Harvest downloader) may have some sort of hard-coded way for dealing with such a case. On the other hand, with the File Harvest downloader, the alternate file name and path can be set by the program for better duplicate file control.

Section 3: Design

At a high level, the File Harvest downloader, or mass downloader, is a work queue of downloads that are to be downloaded by a set of downloaders. The mass downloader coordinates the downloaders and also forwards downloader events up to the user interface as well as implementing some of its own events. Furthermore, the mass downloader controls the starting, pausing, resuming and canceling of the downloaders and their associated downloads.

A mass downloader is created by the developer using the mass downloader class. The developer must also create the downloads and the individual downloaders. In the case of File Harvest, the downloads are created from the downloadable URLs found by the crawler. Once done, the developer can start the mass downloader with the downloads they want downloaded and with the downloaders they want to use to download the downloads. This approach is taken so that the developer can use either the standard downloaders and downloads or they can implement their own according to the IDownload and IDownloader interfaces. First though, let us take a look at the IMassDownloader interface.

```
interface IMassDownloader
{
        event MassDownloaderCompletedEventHandler MassDownloaderCompleted;
        event DownloadCompletedEventHandler DownloadCompleted;
        event DownloadStartedEventHandler DownloadStarted;
        event DownloadProgressChangedEventHandler DownloadProgressChanged;

        bool Paused { get; }
        short Progress { get; }

        void Start(IEnumerable<IDownloader> downloaders,
IEnumerable<IDownload> downloads);
        void PauseAll();
        void ResumeAll();
        void CancelAll();
}
```

Figure 26 - The IMassDownloader interface.

The mass downloader serves many important purposes. For instance, it serves as a centralized source of progress data on a batch of downloads. To this end, it has the Progress property, which can be used to retrieve the overall progress of the current batch of downloads. Calling Start clears the current progress. To avoid clearing the progress,

use the ResumeAll function, which can be used anytime the IMassDownloader is Paused. To clarify, you can not Resume Canceled downloads and you can not Resume downloads that are currently downloading/Started. Also, the events that start with "Download" are actually events from an IDownloader that the IMassDownloader manages. This design decision means that all of the events a client receives come from an IMassDownloader, rather than from the IDownloader; even if some of those events originate from other user-instanced classes.

The next interface that is important to define is the IDownloader interface. An IDownloader is responsible for actually downloading an IDownload and reporting its progress to whomever is using it – in this case, the mass downloader and ultimately the user interface. The following figure shows the IDownloader interface.

```
interface IDownloader
{
        event DownloadStartedEventHandler downloadStarted;
        event DownloadProgressChangedEventHandler downloadProgressChanged;
        event DownloadCompletedEventHandler downloadFinished;

        bool Paused { get; }
        bool Busy { get; }
        IDownload myDownload { set; }

        void Download();
        void DownloadAsync();
        void Pause();
        void Resume();
        void Cancel();
}
```

Figure 27 - The IDownloader interface.

Again, the IDownloader has the ability to be Paused, Resumed and Canceled. Also, there are condition variables to see if the IDownloader is currently Paused or Busy

(downloading). Next, the IDownloader's current IDownload can be set through the myDownload property. Lastly, and most importantly, the IDownloader can be started synchronously or asynchronously.

The next figure contains the IDownload interface.

```
interface IDownload
{
        int Progress { get; }
        long FileSize { get; set; }
        int TotalBytesRead { get;  set; }

        bool Completed { get;  set; }
        bool Canceled { get;  set; }
        bool CanResume { get;  set; }

        Uri Uri { get; }
        string DownloadLocation { get; }
}
```

Figure 28 - The IDownload interface.

The IDownload class is viewed as a data container more so than a class that does work. In fact, an IDownloader does all of the work and actually updates the status of each IDownload. One reason for this is to easily allow an IDownload to be serialized so that it can be saved if the program is closed and then it can be resumed when the program starts next. Although this feature is not implemented, it alone is a strong case for separating the responsibilities of the IDownload and the IDownloader as has been done.

The IDownload stores its Progress, its FileSize, and the number of bytes that have been read so far (TotalBytesRead). The latter two properties actually determine the Progress of the IDownload – it is a derived property.

The IDownload also knows if it is Completed or Canceled and also knows if it CanResume. Some web sites do not allow certain types of HTTP request headers – those that facilitate resuming a download from an arbitrary point in the file – and so the CanResume Boolean ensures that if an IDownload can not be resumed, that the download is started over completely as dictated by the web server.

The last two members of the IDownload interface include the Uri field, which indicates the URL to download, and the DownloadLocation string that indicates where the file should be downloaded to on the local machine.

Section 4: Parallel Downloading

The user of File Harvest is allowed to decide at most how many downloads are being simultaneously downloaded. This implies that downloads typically occur in parallel. To this extent, File Harvest launches a new downloading thread for each download so long as the maximum number of simultaneous downloads has not yet been reached. This feature is also nice because it allows long running downloads to run for as much time as they need, while still allowing other downloads to start and finish.

Thread synchronization is a minor problem as a result of this parallel downloading model. However, the only implication of this decision thus far has been the synchronization of the starting of downloads, which ensures the enforcement of the maximum number of simultaneous downloads.

Section 5: Mass Downloader Client Events

As a user of the Mass Downloader library, File Harvest is able to hook into events that the Mass Downloader exposes to its clients. These events tend to center around the progress of downloads, but also serve other purposes. The following table summarizes these events:

IMassDownloader Events	
Event Signature	**Description**
void MassDownloaderCompletedEventHandler(object sender, MassDownloaderCompletedEventArgs e);	Fires when all downloads have completed.
void DownloadStartedEventHandler(object sender, DownloadStartedEventArgs e);	Fires when an IDownload starts to download or when it resumes downloading.
void DownloadCompletedEventHandler(object sender, DownloadCompletedEventArgs e);	Fires when an IDownload finishes downloading, is cancelled, or is paused.
void DownloadProgressChangedEventHandler(object sender, DownloadProgressChangedEventArgs e);	Fires when progress is made in downloading an IDownload.
void DuplicateFilenameEventHandler(object sender, DuplicateFilenameEventArgs e);	Fires when a filename conflict is detected.

Table 13 - Summary of Mass Downloader events.

Each IMassDownloader event sends not only the IMassDownloader who is firing the event (object sender) but also an object of a type that inherits from the standard .NET EventArgs class. In all events, except the MassDownloaderCompleted event, the EventArgs contain the IDownload associated with the event—allowing the client to retrieve details about that IDownload such as its progress or its file size. Overall, these events, except the DuplicateFilename event, notify the client of the IMassDownloader's progress.

5.1: Duplicate Filename Renaming

The DuplicateFilename event fires before an IDownload starts if the download location of the IDownload conflicts with a file already on the user's computer or if the

download location conflicts with another IDownload that is already downloading. By default, the IMassDownloader will rename the new IDownload so that it no longer conflicts. However, the DuplicateFilename event allows the client to modify the behavior of the IMassDownloader on a per duplicate file basis. There are four strategies the client can make the IMassDownloader use:

Duplicate Filename Rename Strategies	
Strategy	**Description**
IgnoreNew	Cancels the new file download and leaves the original file alone.
RenameNew	Renames the new file so that it does not collide with the original file or any other files in the same directory. To control how a file is renamed, set the newFilename property of the DuplicateFilenameEventArgs or the default renaming scheme will be used. This is the default strategy.
RenameOld	Renames the original file so that it does not collide with the new file or any other files in the same directory. To control how a file is renamed, set the newFilename property of the DuplicateFilenameEventArgs or the default renaming scheme will be used.
OverwriteOld	Replaces the original file with the new file.

Table 14 - Listing of duplicate filename rename strategies.

In addition to setting this property of the DuplicateFilenameEventArgs, the client can also set the newFilename property of the DuplicateFilenameEventArgs. This property is only used for the RenameNew and RenameOld strategies. If the user does set the RenameNew or RenameOld strategy but does not set the newFilename property, the IMassDownloader will use its default renaming strategy on the new or old file respectively. In the case of File Harvest's Mass Downloader, the default strategy is to prefix the filename with "fh_" and suffix the filename (before the file extension) with "_#"—where '#' is a number such as 1, 2 or 3. This number is incremented until the filename no longer conflicts with other filenames.

CHAPTER 6 - FUTURE WORK

Section 1: Crawling

The advanced crawling features of File Harvest are what set it apart from other related applications. Thus, continuing to focus of even more advanced crawling features will continue to keep File Harvest desirable and cutting-edge. To that extent, the following future work in crawling is considered a top priority.

1.1: Protocol Monitoring – Adding More Protocol Support

The first, and most important, concern going forward for File Harvest is its lack of support for popular Internet protocols besides HTTP. Other protocols, such as RTMP, RTSP, and MMS are being used online to deliver media, but File Harvest is unable to capture these types of media because it is unaware they are even being served on the web page.

Section 2: Downloading

2.1: Cleaner Definition of IDownloader and IDownload Interfaces

Although the IDownloader and IDownload interfaces currently suffice for File Harvest's needs, these two interfaces are inherently very connected and interwoven. To help have more understandable and re-usable code, it would be nice to re-factor the interfaces so that they could be more easily decoupled in the actual program logic. This feature is not high on the priority list, but would be a nice side project or something to very slowly, but surely redevelop.

Section 3: Filters

3.1: Smart, Automatic Generation of Filters

One problem users face is having to always set filters and configurations before actually starting a crawling and downloading session. To cut down on the user's time, it would be helpful to have an advanced system that could analyze a web page and suggest optimal session settings – possibly based on the user's past downloading behavior. For instance, if the user only ever downloads videos, the generator would try to find videos on the base web page and create a filter to only look at those URLs.

In a simple view, common configurations could have shortcuts to change all of the program settings to a common theme. For instance, there could be a button that says something such as "2 Levels of Movies & Images", which would change the settings to search to a depth of 2 and to also only search for movies and images on those web pages. These types of shortcuts are more realistic than the idealized automatic generator and are easier to implement. Of course, finding a convenient space for such buttons (or controls) would prove to be the challenge. Also, a button to load the past two configurations or some sort of configuration history would be helpful. As a common theme, developing a way to easily summarize the current configuration in a text or image format is important.

Section 4: User Interface

4.1: Main Window

It goes without saying that the friendliness and usability of the main user interface window is of the up most importance to users. This is even more so true because configuring a crawling and downloading session can be time consuming, so making it as

easy as possible to configure a session will help users use the program more often and also explore its more advanced features, which will also help them as they continue to use the program.

It would be helpful to have the main window more dynamic is style. This means that it might use animations or have a sleeker layout. Additionally, finding a way to have shortcuts to commonly used settings (like search depth) is very important. Of course, knowing when to stop adding shortcuts is the difficult part.

Overall, testing should be done with independent users to analyze how they use the software, what problems they encounter, and what are their likes and dislikes of the program. Using this information, a cleaner and more helpful user interface – especially the main window – can be built.

Section 5: Miscellaneous

5.1: Pattern Generator

The File Harvest Patterned URL Generator is a very powerful tool. Unfortunately, it currently only supports creating wildcards that are integers. It would be helpful to also allow dates or alphanumeric characters to be used to fill in the wildcards. This feature is important as there are very few other good ways to generate many patterned URLs.

5.2: Save Session Settings on a Collection Basis

Similar to reducing configuration time, it would be extremely helpful to save the configuration settings used for a collection and always use those settings for that collection. This would basically mean that the user would only have to configure settings

for a collection of URLs once – unless, of course, the structure of those pages change or the content the user desires from those pages change.

The main problem with this approach is questioning whether or not it is too fine-grained of a feature. Consider that File Harvest is about massive downloading. In other words, users use File Harvest because there is a large amount of media they want to download automatically. Although the user cares about each file they ultimately have downloaded, they may not want control over each and every download as it may simply take too much time to configure. Thus, storing configuration settings on a collection basis would probably allow the user enough control while not forcing them to do massive configuration as well.

5.3: Site Interaction

One reason why a user may not want to use File Harvest is because it completely removes the browsing experience. Take for instance media on a social networking site. Users may want to download a lot of pictures of their friends or several videos, but they also want to comment on those videos and pictures. File Harvest currently does not allow the user to interact with the web pages that are being crawled.

In the future, users may want to be able to pause the crawling process on a certain web page and just explore it before starting to crawl again. These types of features would probably be most easily done through a web browser plug-in that would allow users to see which page is currently being crawled and to optionally pause and resume the crawler from their browser.

Bibliography

[1] Gantz, John F., et al. "The Diverse and Exploding Digital Universe: An Updated Forecast of Worldwide Information Growth Through 2011." Mar. 2008. EMC., Framingham, MA. 27 Mar. 2009. <http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf>.

[2] "US Leisure Time Plummets 20% in 2008, Hits New Low." MarketingCharts. Dec. 2008. Harris Interactive. 27 Mar. 2009. <http://www.marketingcharts.com/television/us-leisure-time-plummets-20-in-2008-hits-new-low-7470/harris-interactive-harris-poll-hours-available-leisure-week-december-2008jpg/>.

[3] "Internet Users Spending Even More Time on Web." eMartketer. 19 Jan. 2009. Harris Interactive. 27 Mar. 2009. <http://www.emarketer.com/Article.aspx?id=1006869>.

[4] “Real-Time Messaging Protocol (RTMP) specification.” Adobe Macromedia. 5 May 2009. <http://www.adobe.com/devnet/rtmp/>.

[5] “AJAX Hello World.” 2005. Dynamic AJAX.com. 6 May 2009 <http://www.dynamicajax.com/fr/AJAX_Hello_World-.html>.

[6] “Designing Secure ActiveX Controls.” 2009. Microsoft Developer Network (MSDN). 6 May 2009. <http://msdn.microsoft.com/en-us/library/aa752035(VS.85).aspx>.

[7] “Flash in HTML.” W3C. 21 Apr. 2009. <http://www.w3schools.com/flash/flash_inhtml.asp>

[8] “XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition).” W3C. Aug. 2002. W3C. 21 Apr. 2009. <http://www.w3.org/TR/html/>